Floating Point Arithmetic

Burt Hashizume
POB 447
Maynard MA 01754

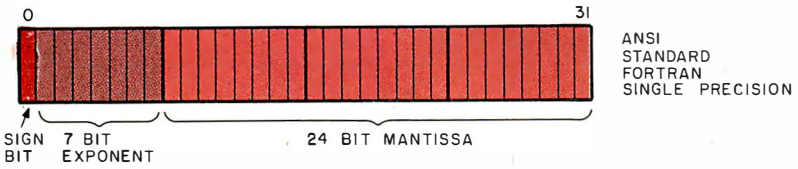


Figure 1: The American National Standards Institute (ANSI) floating point format for FORTRAN. It consists of a 24 bit mantissa, a 7 bit exponent and a sign bit.

Many computer hobbyists are finding 8 bit integer arithmetic inadequate for a variety of mathematical applications. 16 and even 32 bit fixed point calculations are being used with increasing frequency because of their greater accuracy. However, these techniques are still inherently inadequate for calculations performed over a wide range of numbers.

Using a 16 bit integer format, only numbers from 0 to 65,535 can be represented. Larger or smaller numbers can be represented by moving the implicit radix point, but the range of discrete values still remains constant. The fractional part of the quotient in a division of one large number by another could be lost.

If one could dynamically slide the radix point, the number range would be dramatically increased. Using the same format, very small fractions and very large integers can be represented as floating point numbers. This is made possible by keeping track of the radix point's position separately with an exponent.

Floating Point Formats

There are many ways to represent floating point numbers, but there are only three basic formats; the others are variations. Two of these (the dominant ones in the traditional computer industry) use different binary representations. The third format, the one with the most variations, uses a binary coded decimal (BCD) representation, and is widely used in the electronic calculator and home computer industry.

The first format, shown in figure 1, is used in American National Standards Institute (ANSI) FORTRAN. It consists of a 24 bit mantissa, a 7 bit exponent and a sign bit.

The mantissa represents a fraction with the radix point assumed to be to the left of the most significant digit. The exponent is in excess-64 notation, which is a 7 bit two's complement notation with the sign bit inverted, eg: a zero exponent (16^0) is 100 0000, the minimum exponent (16^{-64}) is 000 0000, and the maximum exponent (16^{-63}) is 111 1111. The algebraic sign bit of the value is associated with the mantissa, and the exponent's sign is inherent in its format: a one sign bit indicates the number is negative, and a zero sign bit indicates a positive number.

This is the data storage format of floating point numbers. All such data is assumed to be normalized (ie: the most significant digit in the mantissa is nonzero unless the number itself is zero, in which case all 32 bits are zero). Before a calculation, the numbers are assumed normalized; after a calculation they are normalized in the floating point accumulator before being stored.

The actual calculations take place in the floating point accumulator and other floating point registers. These registers can be in the hardware or in memory (software). Hardware floating point registers (expensive, but much faster than software) are used by large computers and many minicomputers, whereas most small computers implement floating point in software to keep costs down.

With the ANSI format a "guard byte" is used in the floating point registers to maintain accuracy in performing the calculations. The guard byte (see figure 2) is an 8 bit extension to the least significant end of the 24 bit mantissa, temporarily creating a 32 bit mantissa during calculations. By keeping track of 32 bits of accuracy throughout the operation, significance will not be lost when storing numbers because the 32 bits can be rounded off to 24 bits. If a guard byte is not used, no rounding off is possible, and the effect would be the same as truncation (which can result in loss of accuracy very quickly, as will be shown later).

Numbers from 1.00×16^{-65} to $F.FFFFF \times 16^{+62}$ can be represented by this format, resulting in an approximate range of from 10^{-79} to 10^{+76} with an accuracy of six or seven decimal digits. Table 1 lists several decimal numbers along with their hexadecimal ANSI FORTRAN format equivalents.

The next format, shown in figure 3, is also a binary format and is implemented by Digital Equipment Corporation (DEC) and Hewlett-Packard in their BASIC interpreters. It consists of a 23 bit mantissa plus a "hidden bit," an 8 bit exponent and a sign bit.

This format assumes that the number is always normalized. Therefore, the most significant bit (MSB) of the mantissa is always one unless the entire number is zero. If the number is zero, (indicated by the special case of a 0 exponent) then the hidden bit is also zero. The sign bit is zero for a positive number and one for negative. Because all nonzero numbers have an MSB of one, it need not be explicitly represented in the format; hence only 23 bits in the mantissa.

The exponent represents a power of two in excess-128 notation, which is similar to excess-64 notation. The largest exponent, 2^{+127} is represented by the largest number, 1111 1111, and the smallest exponent, 2^{-127}, by the smallest nonzero number, 0000 0001. An exponent of zero (2^0) is represented by 1000 000, while the number zero is reserved to indicate a zero mantissa.

As in the case of the first binary format, a guard byte must be used during calculations so that round off is possible before returning from the floating point accumulator for storage in memory. In this format it is also necessary to explicitly represent the hidden bit during calculations. This is accomplished by expanding the 4 byte format

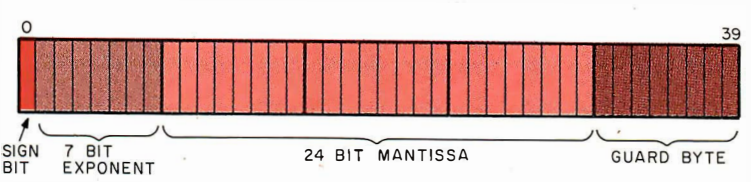


Figure 2: The ANSI FORTRAN floating point format showing the location of the "guard byte." The guard byte is an extra field which holds portions of intermediate calculations so that the final calculated value can be rounded off rather than truncated prior to further use.

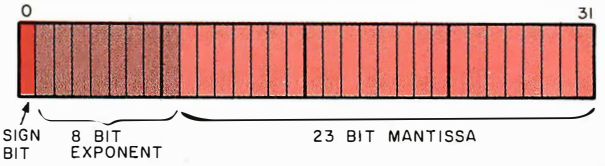


Figure 3: A binary floating point format used by Digital Equipment Corporation and Hewlett-Packard in their BASIC interpreters. It consists of a 23 bit mantissa with a "hidden" bit, an 8 bit exponent and a sign bit. The format assumes that the number to be represented is always normalized: the most significant bit of the number is always understood to be 1 unless the entire number is equal to 0. This assumed "1" bit is the so-called "hidden" bit.

Decimal Number	Hexadecimal Floating Point Number (Hexadecimal Digits)
1.00	41 100000
6.00	41 600000
−1.00	C1 100000
0.50	40 800000
−0.50	C0 800000
100	42 640000
2^{16} (= 65,536)	45 100000
2^{-16}	3D 100000
-2^{-32}	B9 100000
0	00 000000
16^{-65}	00 100000
16^{+62}	7F 100000

Table 1: Several decimal numbers along with their ANSI FORTRAN floating point hexadecimal format equivalents (see figures 1 and 2).

Decimal Number	Binary Floating Point Number (Hexadecimal Digits)
1.00	40 800000
6.00	41 C00000
−1.00	C0 800000
0.50	40 000000
−0.50	C0 000000
100	43 C80000
2^{16}=65,536	48 800000
2^{-16}	38 800000
-2^{-32}	B0 800000
0	00 000000
2^{-128}	00 800000
2^{+126}	7F 800000

Table 2: Examples of decimal numbers and their equivalents as encoded in the binary floating point format used in several BASIC interpreters (see figure 3).

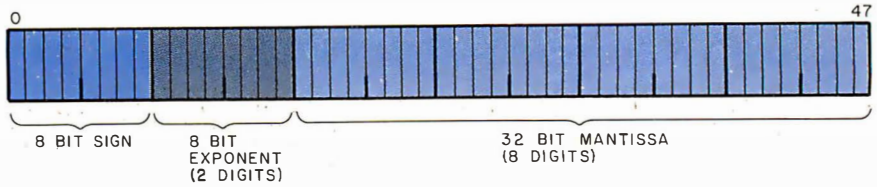


Figure 4: A BCD floating point format consisting of an 8 bit sign, an 8 bit exponent and a 32 bit (8 digit) mantissa.

into six bytes: one byte for the sign, one byte for the exponent, and four bytes for the mantissa (including the guard byte). As a result there is a fair amount of processing necessary to load and store the floating point registers.

This format has a range of from 2^{+126} to 2^{-128} or from approximately 10^{+38} to 10^{-38} with a 7 decimal digit accuracy (several examples are represented in table 2).

There are numerous BCD floating point formats currently in use. Mantissas range from as few as four digits to as many as 16 digits of accuracy, and exponents can typically range from 10^{+99} to 10^{-99}, or even 10^{+127} to 10^{-127}. The most popular format (see figure 4) has an 8 digit mantissa (four bytes of two digits per byte) with the decimal point assumed to be to the left of the most significant digit.

The mantissa sign is typically represented by a whole byte: 00 for positive and 0FF for negative. A variety of formats use one byte to represent the exponent.

One of the more frequently used formats is binary in the form of excess-128 notation. The exponent format itself is identical to the Digital Equipment Corporation format discussed earlier, but represents a power of ten instead of a power of two. Thus, an exponent of 84 base 16, using DEC's format, signifies two to the fourth power, and using the BCD format, ten to the fourth. The exponent represents the same power in both cases, but of different bases.

Eight digits are packed into four bytes in what is known as packed BCD (four bits represent one BCD digit).

The same format is usually used for both storage of data and actual calculations. This means neither a guard byte nor round off is used. The need for a guard byte is circumvented by using more significant digits than are actually necessary, eg: calculating to eight digits for 6 digit results, or calculating to nine digits for 8 digit results. This of course makes it necessary to use more memory per number for storage.

Some examples of numbers in this format are found in table 3.

Table 3: Several decimal numbers along with their equivalent floating point representations as encoded in BCD hexadecimal digits.

Decimal Number	BCD Representation (Hexadecimal Digits)		
1.00	00	81	10000000
6.00	00	81	60000000
−1.00	FF	81	10000000
0.50	00	80	50000000
−0.50	FF	80	50000000
100	00	83	10000000
2^{16}=65,536	00	85	65536000
2^{-16}	00	7C	15258789
-2^{-32}	FF	77	23283064
0	00	00	00000000
10^{+126}	00	FF	10000000
10^{-128}	00	01	10000000

Format Pros and Cons

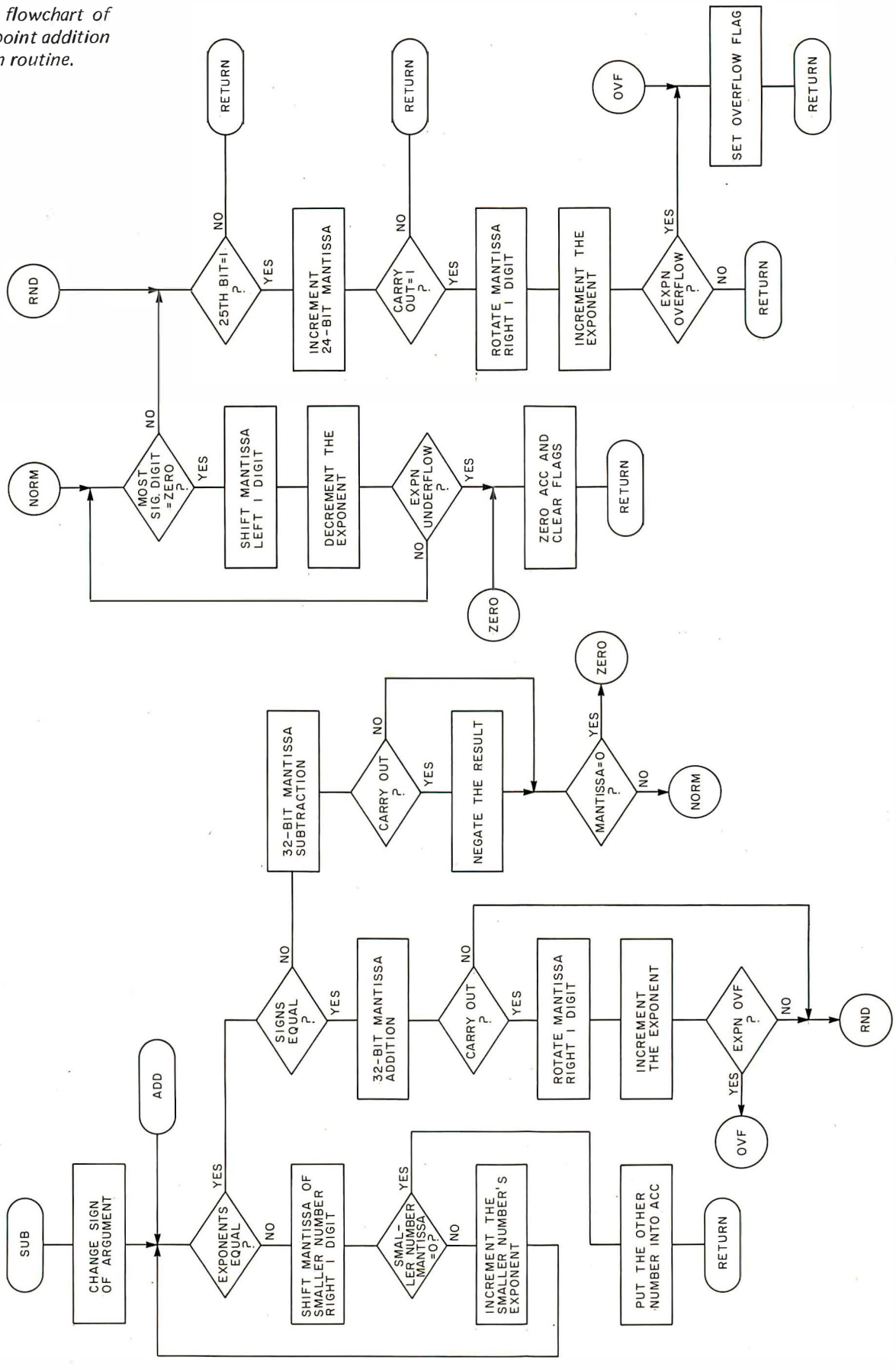
Each of these basic floating point formats has its own particular advantages and disadvantages. Which format is best is dependent upon the requirements of the particular application: speed, small memory size, variable mantissa length, ease of coding in a given computer architecture, ease of interfacing to other software routines, etc.

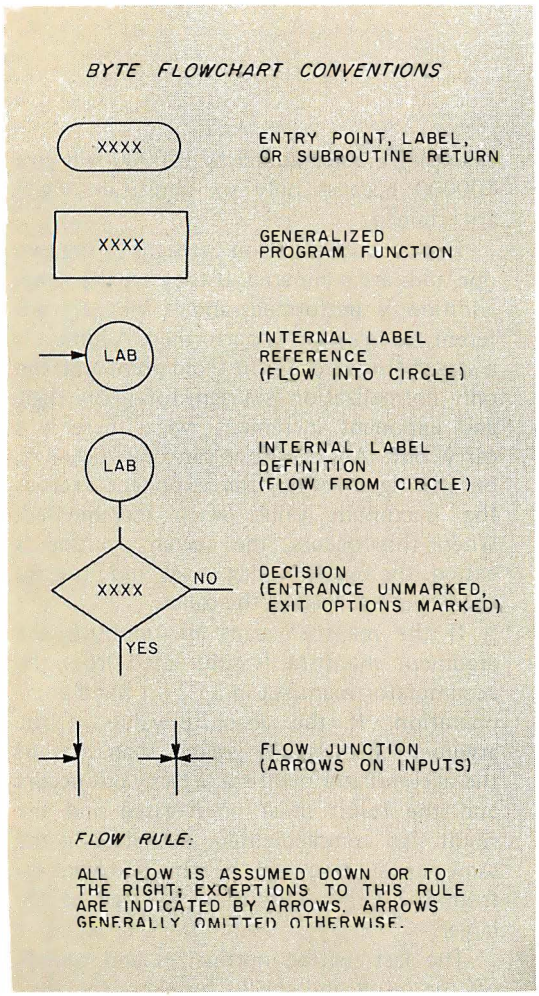
The BCD format with its variations is by far the most popular in the personal computing field, probably because it is the easiest to program. The relative ease in converting from an ASCII representation of a number to the BCD format and back is a key factor, as is the ease with which the

Continued on page 180

Continued from page 78

Figure 5: A flowchart of the floating point addition or subtraction routine.





number of digits can be expanded.

This format has its list of disadvantages, though; but for these the commercial computer industry might have adopted it long ago. The program size required for performing just the basic operations and the conversion routines is about the same as for the other formats, but execution times are significantly slower. Many hobbyists are not as concerned with the number of milliseconds as with the number of bytes, but another disadvantage is the larger memory required to store the floating point numbers. For most assembly language applications the impact is negligible. It does become noticeable, however, when the floating point package is part of higher level language programs such as interpreters or compilers. One major disadvantage is more subtle. Many of the transcendental functions are best implemented using algorithms which are binary based. Using these algorithms, the BCD format is awkward at best and at worst consumes large quantities of time and memory.

The binary floating point format provides the fastest execution times, despite the fact that its format allows representation of 7 digit numbers at all times. Because the entire format is in binary, implementing the basic operations and all of the transcendental functions is easier than when using either of the other two formats.

The major drawback is the small range of numbers representable, relative to the other formats (10^{+38} to 10^{-38}). This is because its exponent is only a power of two compared with bases of 10 and 16 respectively. Two other minor drawbacks are the need for routines to convert floating point numbers from a decimal base to a binary base (and vice versa), and the need to expand the binary format to perform actual calculations.

The hexadecimal floating point format permits a much larger number range (10^{+76} to 10^{-76}) than the binary format, and the conversion routines are similar for both. Although slightly slower than the binary format, the hexadecimal format is still much faster than any BCD format of comparable capability.

It is somewhat more difficult to implement scientific functions such as square root, exponential and logarithm with this format than with the binary format, and its precision is not as great as the binary format's precision because it is digit rather than bit oriented. Even though the most significant digit is nonzero, the most significant three bits of the digit itself may be zeroes, resulting in only 21 bits of accuracy. This translates to only six digits of accuracy.

In describing the four basic floating point operations and the format conversions, the hexadecimal format will be used to illustrate examples.

Floating Point Operations

The software uses three floating point registers, an accumulator, argument register and scratch register. The floating point accumulator contains one of the operands prior to a calculation, and the result after the calculation is performed. The argument register contains the other operand, which is loaded by the routine, and the scratch register is used to hold temporary results.

In each of the basic operations there are two parts: exponent calculation and mantissa calculation. Fixed point operations require only the mantissa calculation, which turns out to be the easier of the two.

Add and Subtract Routine

Figure 5 is a flowchart of the add and subtract routine. The two operations are described together because the algorithms

$$
\begin{aligned}
A &= .100000 \times 16^{1} \\
B &= .FFFFFF \times 16^{0}
\end{aligned}
$$

Figure 6a: Two numbers A and B, which differ from one another by less than one part in 2^{24}, but which were represented as two different numbers.

	Mantissa	Guard Byte
A =	.100000	00 × 16^{1}
B =	.0FFFFF	F0 × 16^{1}

Figure 6b: The same numbers as figure 6a, but with B shifted to the right one digit, and the extra digit stored in the guard byte in preparation for the subtraction shown in figure 6c. This shifting aligns mantissa radix points (makes exponents equal).

	Mantissa	Guard Byte		
A =	.100000	00 ×	16^{1}	
−B =	−.0FFFFF	F0 ×	16^{1}	
C =	.000000	10 ×	16^{1}	= .100000 × 16^{-5}

Figure 6c: The subtraction of B from A to give C. There is only one significant digit in the result, which is entirely located within the guard byte.

A =	.100000 ×	16^{1}	
−B =	−.0FFFFF ×	16^{1}	
C =	.000001 ×	16^{1}	= .100000 × 16^{-4}

Figure 6d: If the guard byte is omitted, as in this example, the apparent result is off by a factor of 16 due to truncation prior to the mantissa addition (or subtraction).

are identical except for a sign change before executing a subtract.

The add and subtract routine consists of three functionally separate sections. The first prepares the numbers for the operation by aligning the radix points. This is analogous to aligning the decimal points for an addition or subtraction of decimal numbers. The addition or subtraction is then performed and the result normalized.

The radix points are aligned by shifting the mantissa of the smaller number right one digit and incrementing its exponent until the exponents are equal. When shifting right, the last eight bits shifted out are saved in the guard byte in order to maintain accuracy. During the shifting and incrementing loop, the 32 bit mantissa, including the guard byte, should be checked for all zeroes (a situation which implies that one operand is too small to affect the other). This is to avoid shifting insignificant zeroes. For example, 0.0001 added to 100000 will give 100000 because only six significant digits are retained.

In the second section the signs of the two operands are compared. If they are the same, addition is performed, and if they are different, subtraction is performed. Addition is a straightforward 32 bit fixed point add; the only normalization is a right rotate one digit and exponent increment when there is a carry out. An overflow can only occur if, on the right rotate, the exponent exceeds the maximum value when incremented. When this occurs, the current routine is exited, the overflow flag is set, and program control is returned to the caller.

If the mantissa signs are opposite, the argument mantissa is subtracted from the accumulator mantissa in a 32 bit fixed point operation. If the absolute value of the argument mantissa is greater than that of the accumulator mantissa, a carry out occurs and the result must be negated and the result sign complemented. The effect is the same as subtracting the smaller mantissa from the larger and using the sign of the larger.

The last section normalizes and rounds off the result and checks for exponent overflow and underflow. Normalization consists of shifting the mantissa digits left until the most significant digit is nonzero. For each shift, the exponent must be incremented and checked for overflow. Only 24 bits of mantissa are saved. Therefore, the 25th bit of the temporary result determines whether the mantissa is to be rounded up or not. For example, if the hexadecimal result were 10000094, it would be rounded up to 100001, whereas a result of 10000048 would not.

If the guard byte and a round off operation are not used in an addition, one bit of significance could be lost. By comparison, subtraction without a guard byte could mean a difference of an order of magnitude. Two numbers can be different by less than one part in 2^{24} and yet be represented as two different numbers (A and B in figure 6a). When one is subtracted from the other, the smaller must be shifted right in order to align the radix points. The guard byte stores the shifted out digit (figure 6b) and retains the only significant digit of the result (figure 6c). Without a guard byte the significant digit may be off by a factor of 16 (figure 6d).

IF YOU'RE NOT SUBSCRIBING TO CREATIVE COMPUTING, YOU'RE NOT GETTING THE MOST OUT OF YOUR COMPUTER.

No computer magazine gives you more applications than we do! Games. Puzzles. Sports simulations. CAI. Computer art. Artificial intelligence. Needlepoint. Music and speech synthesis. Investment analysis. You name it. We've got it. And that's just the beginning!

Whatever your access to computer power—home computer kit, mini, time-sharing terminal—*Creative Computing* is on your wavelength. Whatever your computer application—recreation, education, business, household management, even building control—*Creative Computing* speaks your language.

Read through pages of thoroughly documented programs with complete listings and sample runs. All made easy for you to use. Learn about everything from new software to microprocessors to new uses for home computers. And all in simple, understandable terms. And there's still more. *Creative Computing* discusses creative programming techniques like sort algorithms, shuffling and string manipulation to make your own programming easier and more efficient.

We can even save you time and money. Our extensive resource section is filled with all kinds of facts plus evaluations of hundreds of items. Including microcomputers, terminals, peripherals, software packages, periodicals, booklets and dealers. We also give you no-nonsense equipment profiles to help you decide which computer is best for you—before you spend money on one that isn't.

We've got fiction too. From the best authors in the field, like Asimov, Pohl and Clarke. Plus timely reviews of computer books, vendor manuals and government pamphlets. And so much more!

Isn't it about time you subscribed to *Creative Computing*? It's the smart way to get the most out of your computer.

Complete this coupon and mail it today. Or for fast response, call our toll-free hot line. (800) 631-8112. (In New Jersey call (201) 540-0445).

I want to get the most out of my computer.
Please enter my subscription to:

creative computing

Term	USA	Foreign Surface	Foreign Air
☐ 1-year	☐ $ 8	☐ $ 12	☐ $ 20
☐ 2-year	☐ $ 15	☐ $ 23	☐ $ 39
☐ 3-year	☐ $ 21	☐ $ 33	☐ $ 57
☐ Lifetime	☐ $ 300	☐ $ 400	☐ $ 600
☐ Vol. 1 Bound	☐ $ 10	☐ $ 12	☐ $ 15
☐ Vol. 2 Bound	☐ $ 10	☐ $ 12	☐ $ 15

☐ Payment Enclosed
☐ Visa/Bank Americard ☐ Master Charge
Card No. _____
☐ Please bill me ($1.00 billing fee will be added; foreign orders must be prepaid)

Name _____
Address _____
City _____ State _____ Zip _____

Send to: Creative Computing, Atten: Inez
P.O. Box 789-M, Morristown, N.J. 07960

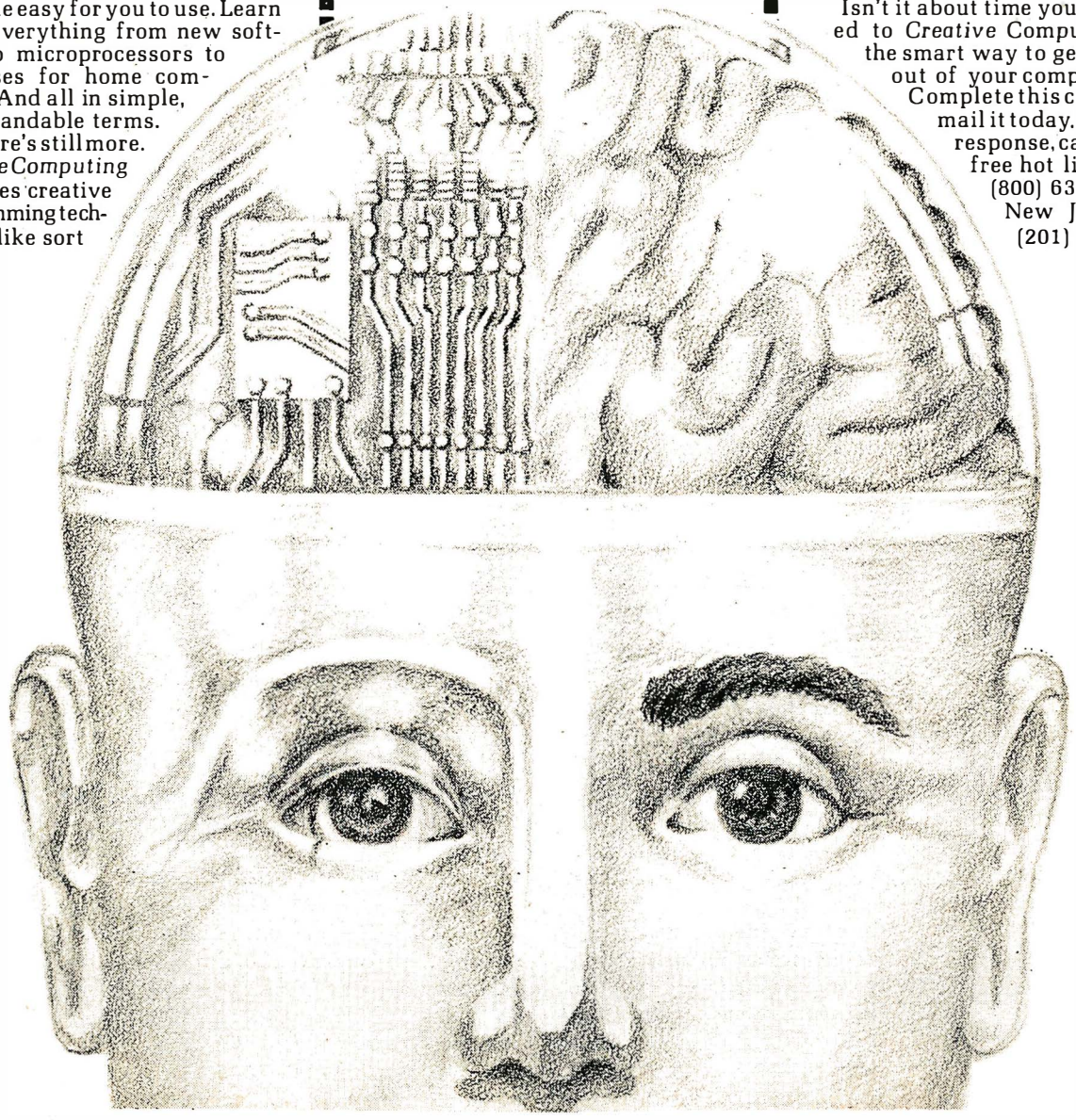
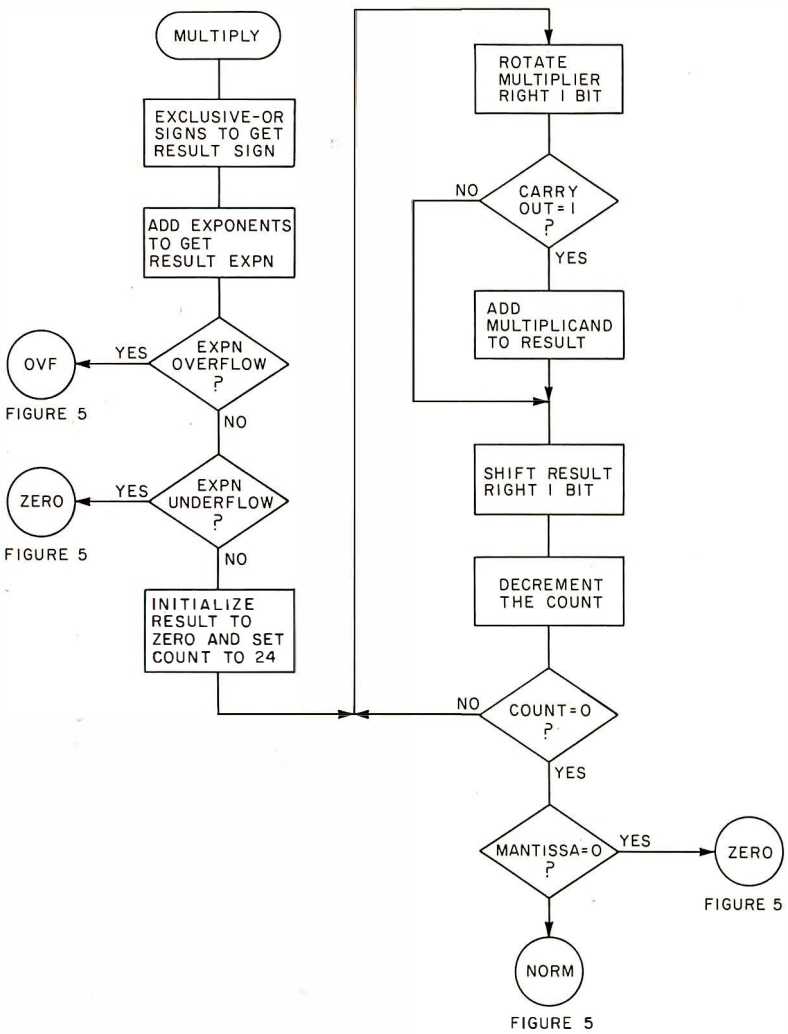


Figure 7: A flowchart for the floating point multiplication routine.



Multiplication

Figure 7 is a flowchart of the multiplication routine. Calculation of the exponent for the multiplication and division routines is achieved by adding or subtracting the operand exponents respectively. Since the exponents are in excess-64 notation, the offset (64) will have to be subtracted from or added to the result. If the resultant exponent is less than the smallest exponent or greater than the largest, an underflow or overflow condition exists and the appropriate action is taken (for example, displaying an error message or setting the result to a fixed value). Sign calculation for both multiply and divide is a simple exclusive or of the two operand signs.

The partial product method is the most widely used in fixed point multiplications, decimal or binary based. Using binary numbers, this algorithm rotates the multiplier right one bit and tests the bit rotated out. The multiplicand is conditionally added to the accumulated result if the bit is a one. The result is then rotated right one bit, retaining 32 bits, and the whole procedure repeated for all 24 bits of the multiplier. *[An example of this algorithm implemented in hardware was found in the article "This Circuit Multiplies" by Tom Hall, page 36 in July 1977 BYTE . . .* CH*]*

Though the fixed point calculation is straightforward and uncomplicated, it is extremely time consuming because the loop is repeated 24 times. One method of reducing the execution time is to cut out all subroutines within the loop and use only in line code. A complete multiplication routine can then have a worst case multiply time of about 2.5 ms using an 8080 processor with 2 MHz clock.

Division

Figure 8 is a flowchart of the division routine. The fixed point divide algorithm is analogous to the partial product method and is also commonly used. It compares the absolute value of the divisor to that of the dividend. If it is equal to or less than the dividend's absolute value, it is subtracted from the dividend, and a one is rotated into the least significant bit of the quotient. Otherwise there is no subtraction and a zero is rotated in. The dividend is then shifted left one bit and the loop repeated for a total of 32 times, generating a 32 bit quotient. Long division by hand goes through the identical procedure, but it operates on digits instead of bits.

Since more processing is done in each loop cycle than in the multiply routine, division execution times are longer than multiplication times. The worst case times are still around 5 ms for an 8080 with 2 MHz clock.

In both the multiply and divide routines, the normalization procedure is identical to the one in the subtract routine. Therefore it usually turns out to be shared code.

These routines are the core for other floating point functions such as format conversions and scientific mathematical functions. Because of this it is important that these routines execute as fast as possible so that the other functions' execution times are not increased to several seconds instead of fractions of seconds.

BCD to Binary and Binary to BCD conversions are probably the most difficult to implement in a binary floating point package. There are several simple methods of converting integers from one format to the other, but I haven't seen any published literature to date on either floating point arithmetic or number base conversions.

The methods described here were chosen because of their simplicity rather than their speed. The slow base conversions are still relatively fast compared to the character oriented input and output operations in which they are used, so for most purposes the conversion speed is not noticeable.

Decimal to Binary Conversion

The Decimal to Binary (DB) routine (figure 9) converts a free format floating point BCD number in ASCII to binary floating point format, converting from ASCII BCD floating point to formatted BCD floating point, and then to binary floating point in one operation.

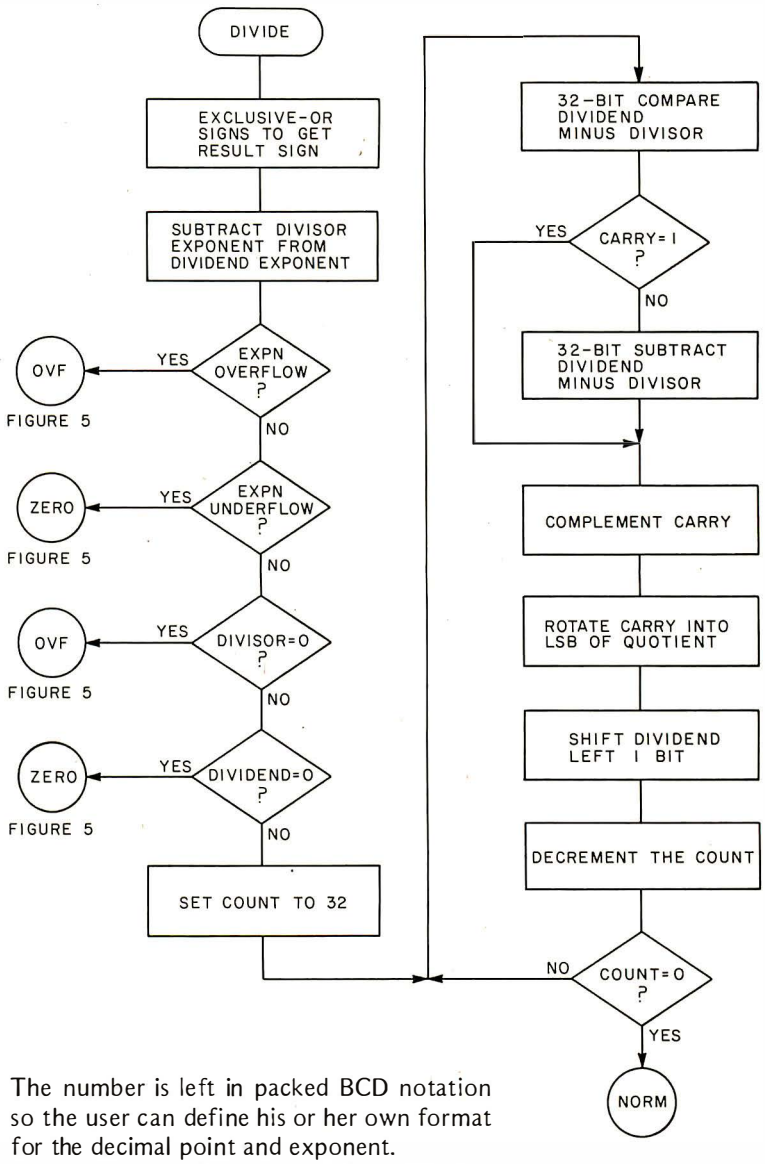
After initialization the DB routine first checks for a plus or minus sign, which is optional. It ignores a plus sign and sets a flag if there is a minus sign. It then reads in one or more digits (and possibly a decimal point). When it encounters a decimal point, it tests a flag to see if another decimal point has already occurred and sets the flag if not. If a decimal point has already occurred, the routine jumps to the last section. For each decimal digit input, the routine multiplies the accumulated result by ten in floating point format, creates a floating point number from the digit, and adds the number to the accumulated result. If a decimal point has previously occurred, a decimal exponent count is decremented, keeping track of the number of digits in the fractional part. This process is repeated until a character which is neither a digit nor decimal point has occurred, at which point control passes on to the exponent evaluation routine.

Here the decimal exponent of the number, if any, is processed. The routine first searches for the presence of an E character. If none is present, control jumps to the last section. If the character is present, one or two BCD digits are inputted with an optional plus or minus sign. The BCD digits are converted to an 8 bit binary, two's complement number and added to the decimal exponent count.

Finally, the mantissa is normalized by either repeatedly multiplying or dividing by ten, depending upon the decimal exponent count. Multiplication is performed if the count is greater than zero, and division is performed if it is less than zero. The count is either decremented or incremented respectively toward zero for every multiplication or division. When the count reaches zero, the sign is corrected if the number is negative, and the routine returns.

The Binary to Decimal (BD) routine shown in figure 10 converts a binary floating point number to packed BCD floating point.

Figure 8: A flowchart for the floating point division routine.



The number is left in packed BCD notation so the user can define his or her own format for the decimal point and exponent.

Initially, the binary number is normalized so that it is in the range of 0.1 to 1.0, with a decimal exponent kept separate. This is done by repeatedly multiplying or dividing by 10 until the number is equal to or greater than 1.0 and less than 10.0, and then dividing it by 10.0. During this operation, each multiplication or division by 10 is tabulated in a count. Next, a round off of 0.0000005 is added and a correction, if necessary, is made to make sure the number remains between 0.1 and 1.0.

The number is then converted to a binary fixed point fraction, and finally to a BCD fixed point fraction of eight digits, but accurate to only six digits because of the added round off.

After completing mantissa conversion, the binary count of the decimal exponent is converted to a signed BCD pair and stored with the BCD fraction.

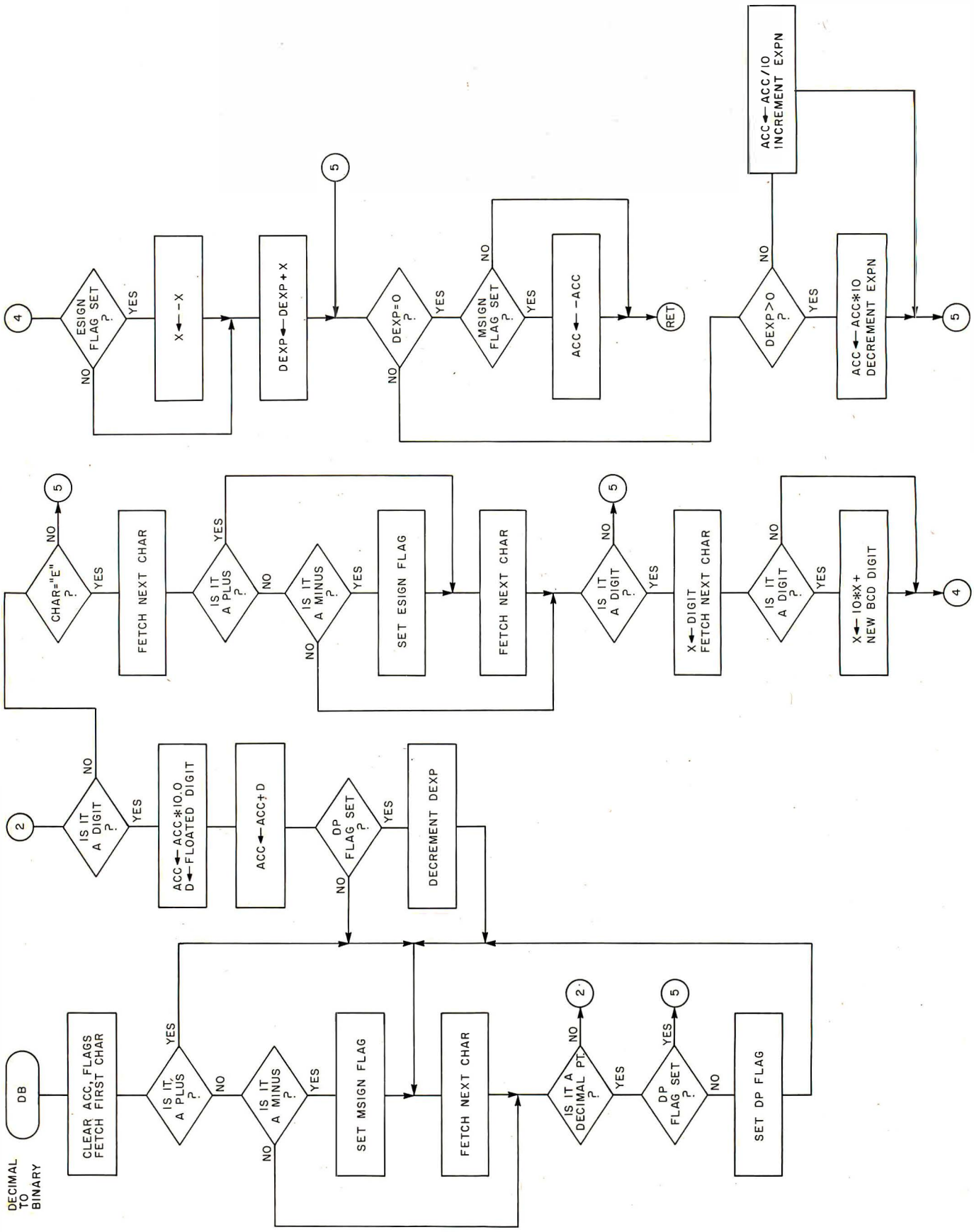


Figure 9: Flowchart of a decimal to binary routine used to convert a free format floating point BCD number in ASCII format to binary floating point format.

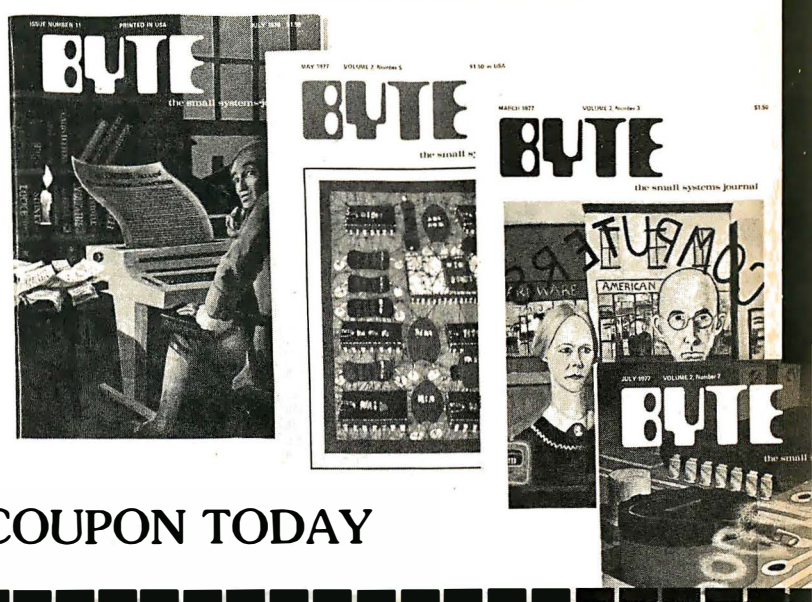
NOTICE
to all BYTE magazine subscribers and would-be subscribers

SAVE MONEY! As of January 1, 1978, we are raising our subscription rates to meet the increased costs of producing and mailing *BYTE* to our more than 110,000 monthly readers. The new domestic subscription rates, effective January 1, 1978:

Current Rates		Rates Effective January 1, 1978	
One year U.S.	— $12	One year U.S.	— $15
Two years U.S.	— $22	Two years U.S.	— $27
Three years U.S.	— $32	Three years U.S.	— $39

If you already subscribe to *BYTE* and still have several months to go before expiration of your subscription, you can still take advantage of current rates for renewal (even up to three years). Use coupon below. When your present subscription expires, your renewal order commences.

If you are not yet a subscriber to *BYTE*, the leading magazine for the creative home computer experimenter, don't delay . . . mail this coupon and start getting your own copies of this invaluable magazine, before new rates become effective.



USE THIS MONEY-SAVING COUPON TODAY

To new subscribers:
Read your first copy of *BYTE*, if it's everything you expected, honor our invoice. If it isn't, just write "CANCEL" across the invoice and mail it back. You won't be billed, and the first issue is yours.

Allow 6 to 8 weeks for processing.

Circle 20 on inquiry card.

BYTE Subscription Dept. • P.O. Box 361 • Arlington, Mass. 02174

PLEASE ENTER MY SUBSCRIPTION FOR:

☐ One year $12 (12 issues) ☐ Two years $22 ☐ Three years $32

☐ Check enclosed (entitles you to bonus of one extra issue)

☐ Bill me ☐ Bill BankAmericard/Visa ☐ Bill Master Charge

Card Number: ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date: _____

Signature: _____ Name *(please print)* _____

Address: _____

City: _____ State/Country: _____ Code: _____

Offer expires December 31, 1977

FOREIGN RATES FOR ONE YEAR: *(Please remit in U.S. Funds)*

☐ Canada or Mexico $17.50 (Air *delivered)* ☐ Europe $25 (Air delivered)

☐ All other countries except above: $25 *(Surface delivery)*

Air delivery available on request

These two algorithms for conversion of bases between BCD (base 10) and binary (base 2) are valid for any binary floating point format, not just the one used here.

Concluding Remarks

It is hoped that this discussion along with the flowchart specifications of the algorithms can be used by readers as a basis for coding a floating point arithmetic package for any general purpose microprocessor system. I have used this information in particular to code an 8080 version of the routines for the basic arithmetic functions, as well as extensions for functions such as square root, exponential, natural logarithm, sine and cosine, and arc tangent. The extensions all use the basic multiplication, division, addition and subtraction operations to evaluate the more complex functions involved. Readers interested in a detailed copy of this 8080 mathematical function software documentation can purchase it for $10 by writing to me at POB 447, Maynard MA 01754. ■

Figure 10: Flowchart of a binary to decimal conversion routine used to convert a binary floating point number to packed BCD floating point format.